# IPF : un langage pour le traitement d'image sur architecture SIMD : définition et performances

Yves Robin, Benoit Guérin, François Bodin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *IPF : un langage pour le traitement d'image sur architecture SIMD. Définition et performances*

Yves ROBIN, Benoit GUÉRIN,
François BODIN

**N° 2159**
Janvier 1994

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

# IPF : un langage pour le traitement d'image sur architecture SIMD : définition et performances

Yves Robin, Benoit Guérin, François Bodin

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet CALCPAR

Rapport de recherche n° 2159 — janvier 1994 — 38 pages

**Résumé :** Cet article présente le langage IPF (Image Processing Fortran) et une première version de son compilateur sur DECmpp 12000. IPF a été élaboré pour les besoins du traitement d'image en temps réel vidéo, sur une architecture SIMD à mémoire distribuée. Doté d'une syntaxe Fortran 90 réduite, IPF modélise les images sous forme de tableaux et les considère comme des objets parallèles. Les spécificités du domaine d'application ont permis de développer des techniques de compilation performantes : le placement des données en mode bloc et la technique d'adressage modulo ont été appliqués efficacement au décalage régulier de tableau. Les premiers résultats comparatifs avec le compilateur DEC HPF version 3.1 sont favorables à IPF et montrent que la spécialisation d'un langage à un domaine d'application permet d'obtenir de bien meilleures performances.

*(Abstract: pto)*

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

# IPF: a language for image processing on massively parallel architecture. Definition and performances

**Abstract:** This paper introduces the IPF language (Image Processing Fortran) as well as a first prototype of the compiler on DECmpp 12000 machine. IPF has been designed for real-time image processing on SIMD distributed memory systems. IPF syntax is based on Fortran 90 standard. Therefore digital pictures are represented and processed as parallel arrays. The application field specificities are taken into account in order to develop powerful compiling techniques. Particularly, the block mode mapping for parallel data allocation and the modulo adressing technique are used for array shifting operations. Compared to DEC HPF compiler version 3.1, our solution turns out to be more efficient. In the framework of image processing, such a dedicated language allows high-level expression of algorithms under real-time constraints.

# 1 Introduction

Dans le cadre du projet P$^3$I (Processeurs Parallèles Programmables pour l'Image), les laboratoires de Thomson-CSF/LER ont développé une machine (Figure 1) dédiée au traitement d'image en temps réel vidéo [Battini 91] [Colaitis 91]. Cette machine s'organise en unités indépendantes contrôlées par un anneau de transputers T800. Elle est composée en particulier d'une ou de plusieurs unités massivement parallèles SIMD à mémoire distribuée. Une unité SIMD de P$^3$I est constituée classiquement d'un réseau de processeurs bidimensionnel rebouclé et d'une mémoire de masse auxiliaire. Elle ne dispose toutefois pas de routeur global.

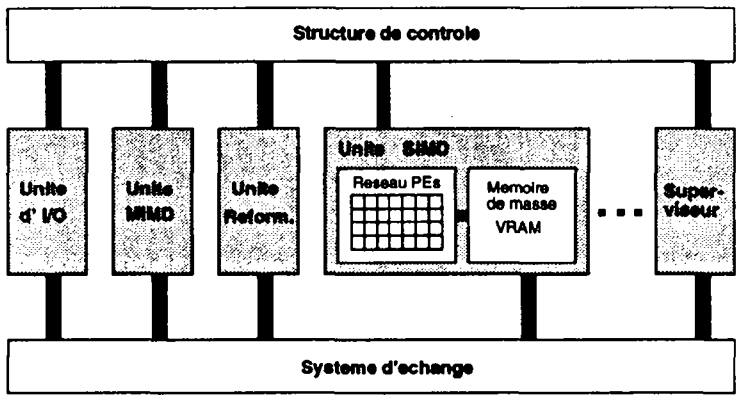


Figure 1 : architecture de la machine P$^3$I

Dans le cadre de ce projet, nous avons décidé de développer un langage de plus haut niveau que le micro-code afin de faciliter la programmation de la partie SIMD de la machine P$^3$I et de réduire les coûts de micro-programmation. Nous avons défini pour cela le langage IPF (Image Processing Fortran) dont nous présentons la syntaxe et le compilateur dans ce rapport. IPF est conçu pour la programmation et la compilation d'algorithmes de traitement d'image en temps réel vidéo sur l'unité SIMD de P$^3$I. Pour valider l'approche haut niveau de IPF et les schémas de compilation destinés à P$^3$I, alors en phase de mise au point, nous avons expérimenté un premier prototype sur DECmpp 12000, dont l'architecture est similaire à celle d'une unité SIMD de P$^3$I. Nous nous sommes seulement contraints à ne pas utiliser le routeur global dont elle dispose. Nous exposons ici une étude comparative entre IPF et le compilateur DEC HPF (High Performance Fortran) version 3.1 disponible sur DECmpp 12000 [DEC 92].

Nous précisons maintenant les données du problème. Compte-tenu de l'importance de l'aspect temps réel vidéo des applications, les contraintes par ordre de priorité sont les suivantes :

- des performances très proches de la micro-programmation manuelle,
- un langage adapté à l'expression du parallélisme dans le cadre du traitement d'image,
- une expression simple des algorithmes de traitement d'image,
- la portabilité à d'autres architectures SIMD.

La première contrainte étant la plus forte, elle a constitué l'un des guides principaux pour la conception du langage. Les performances à atteindre doivent être très proches de celles du langage assembleur de la machine SIMD cible. Pour cela, nous avons exploité les spécificités du traitement d'image pour définir les bases d'un langage qui permette une mise en oeuvre efficace. La question à laquelle nous avions à répondre était la suivante : quel langage assure l'expression aisée des problèmes en traitement d'image et permet une mise en oeuvre efficace sur une architecture SIMD ?

De nombreux langages existants peuvent convenir pour décrire les algorithmes de traitement d'image. Notre choix a été guidé par le besoin d'exprimer le plus simplement le parallélisme dans les programmes, par le souci de limiter le travail de l'utilisateur. Les langages classiques (C, Fortran 77) séquentiels ont été écartés parce que l'écriture des algorithmes est complexe et la détection du parallélisme est difficile : la parallélisation automatique de code séquentiel ne garantit pas un parallélisme maximal et risque de conduire à des performances insuffisantes pour le domaine du traitement d'image temps réel. Ces mêmes contraintes nous conduisent à laisser néanmoins l'opportunité au programmeur d'intégrer des primitives micro-codées qu'exigent les performances vidéo et que ne peut offrir le compilateur.

Le langage spécialisé de haut niveau que nous avons défini dans le cadre du projet, constitue une alternative avantageuse au micro-codage actuel de librairies de traitement d'image et rend plus facile la programmation bas niveau du réseau SIMD de $P^3I$. L'approche haut niveau permet de simplifier de nombreuses étapes lors de la programmation, de réduire les risques d'erreur et les temps de développement. Abstraire les détails architecturaux de la machine sous-jacente permet de faciliter la programmation des algorithmes. La virtualisation des données [Christy 91] par exemple est une technique facile à intégrer dans un compilateur et décharge l'utilisateur de sa gestion. C'est un avantage de l'approche haut niveau. Parmi les

langages parallèles existants qui peuvent convenir à nos besoins, nous distinguons deux catégories :

- Ceux de niveau intermédiaire qui ne gèrent pas la virtualisation des données. Les langages parallèles tels MPL C [Becher 91], Parallel Pcode [Reeves 84], L [Bouge 90], Parallel C [Kuehn 85], Parallaxis [Braun 91] sont proches de la machine. La gestion de la virtualisation est laissée à la charge de l'utilisateur. Les primitives de communication inter-processeurs sont explicites et la portabilité des programmes est réduite.

- Les langages parallèles de haut niveau réduisent le travail de programmation de l'utilisateur, facilitent la lisibilité et la mise au point des programmes. Mais ce sont des langages, soit conçus pour une architecture MIMD (PC++ [Gannon 91], Adapt [Webb 92], Divacon [Mou 90]) ou une machine particulière (CM Lisp [Steele 86], MPP Pascal [Busse 88]), soit élaborés dans l'optique d'une utilisation générale et non pour les besoins d'un domaine d'application précis (C* [Steele 87] [Quinn 91], UC [Bagrodia 90a] [Bagrodia 90b], DAPL [Rice 88], Modula 2* [Philippsen 91], Paralation Lisp [Sabot 88], PompC [Paris 88], Parallel Pascal [Reeves 84] et Fortran 90 [Metcalf 90] [Albert 91]). Ces langages généralistes de haut niveau disposent de fonctionalités que nous n'utilisont pas en traitement d'image. Ces fonctionnalités difficiles à mettre en oeuvre de manière efficace, les pointeurs par exemple, limitent les techniques de compilation exploitables et réduisent ainsi les performances potentielles du langage.

Par contre, la conception du langage IPF et de son compilateur a été soigneusement guidée dans l'optique d'implémenter des algorithmes de traitement d'image, pour des applications temps réel vidéo. IPF ne contient que les fonctionnalités nécessaires à l'expression et à la résolution des problèmes du domaine. La Section 2 présente un survol des algorithmes employés en traitement d'image et adaptés aux machines SIMD à mémoire distribuée. En particulier on s'intéressera dans cette section aux fonctionnalités et structures de données que doit posséder un langage pour permettre une expression simple des algorithmes. La Section 3 présente le langage IPF. Celui-ci s'inspire de Fortran 90, dont nous avons retenu les structures de données et les opérateurs nécessaires au traitement d'image ; seules les constructions permettant une mise en oeuvre efficace ont été considérées. Par contre, des extensions simples de Fortran 90 ont été faites pour assurer une expressivité plus importante et une meilleure efficacité des algorithmes. La Section 4 présente le prototype de compilateur sur DECmpp 12000 et la mise en oeuvre des opérations de gestion des structures de données.

Les opérateurs de base du langage ainsi que quelques algorithmes sont comparés avec le compilateur DEC HPF 3.1 sur DECmpp 12000.

# 2 Traitement d'image et parallélisme massif

Le traitement d'image est un domaine d'application qui englobe des problèmes aussi variés et différents que la restauration d'image, la détection d'objet et la reconstruction de scène tri-dimensionnelle. Tous les algorithmes ne sont pas adaptés à un traitement SIMD. Ils peuvent être classés hiérarchiquement en trois niveaux : algorithmes élémentaires, composés, de haut niveau [Rosenfeld 82] [Duff 86] [Herbordt 91]. Cette classification s'établit selon la structure et le volume des données en entrée, traitées et produites en sortie d'algorithme.

- *Les algorithmes élémentaires* : le seuillage, la FFT [Jamieson 86] [Jacobsen 90a] [Tong 91], le filtrage [Fang 89] [Jacobsen 90b] [Warpenburg 82], la rotation [Strong 82] [Reeves 85], l'extraction de contour [Guerra 87] [Noble 88], la labellisation par *Broadcast* et par *Shrinking* [Cypher 90] [Hsu 90] [Manohar 89] [Nassimi 80] [Hambrusch 84] [Miller 85] [Chaudhary 91], etc. Ils tendent à améliorer la qualité des images et à simplifier leur contenu. Ils manipulent exclusivement des images de pixels et ne modifient pas leur structure bi-dimensionnelle de base.

- *Les algorithmes composés* : la segmentation par *Region growing* [Tilton 88] [Reeves 90] [Strong 91], par *transformée de Hough* [Kannan 89] [Li 93] [Cypher 87], le suivi de contour [Zerubia 92] [Chen 93], l'estimation de mouvement [Mémin 92] [Ibrahim 85], la mise en correspondance [Strong 91] [Williams 86], le *Convex Hull* [Holey 92]. Ils partent d'images pré-traitées et extraient des informations sur les objets qui composent la scène. Le résultat peut être une liste de segments de droite, un centre d'inertie, un champ de mouvement, etc.

- *Les algorithmes de haut niveau* : la stéréoscopie, le flot optique, la reconstruction de surface, etc. Ils ont pour objet d'interpréter la scène contenue dans une séquence d'images et font appel aux traitements de niveau inférieur. Basés sur la mise en relation d'objets, la prise en compte de leurs caractéristiques, de leur localisation, ces traitements permettent de prendre des décision vis-à-vis de l'environnement.

Les algorithmes élémentaires se prêtent bien à une mise en oeuvre SIMD dans la mesure où les traitements sont point à point ou sur un voisinage très restreint. Lorsque le niveau du traitement augmente, le volume d'information à traiter diminue mais les calculs se complexifient alors. Le parallélisme de données potentiel dans ces algorithmes varie donc avec leur niveau et se trouve très limité pour les traitements de haut niveau. Ceux-ci se prêtent mal à une implémentation sur une architecture SIMD où les critères fondamentaux sont un volume important de données homogènes et des opérations simples et uniformément réparties[1]. Des travaux ont été menés afin de déterminer les limites raisonnables d'utilisation d'une telle architecture pour le domaine qui nous intéresse. Willebeek-LeMair et Reeves ont étudié de nombreux algorithmes sur la segmentation par *Region growing* et la représentation des arbres par tableaux [Reeves 90]. Kannan et Chuang ont réussi à traiter le problème de la transformée de Hough appliquée à la détection de segments de droites [Kannan 89] sur architecture SIMD. Li a expérimenté sa généralisation [Li 93]. Mémin et Heitz ont introduit de nouvelles méthodes de relaxation par champs markoviens pour l'analyse d'images qui exploitent les caractéristiques de cette architecture [Heitz 90] [Mémin 92]. Anandan a validé le problème d'estimation de mouvement [Anandan 86] et Miller la détection de formes [Miller 91], directement d'après la structure image de base.

La Table 1 présente un résumé de ces travaux, à travers un échantillon d'algorithmes. Les algorithmes choisis ont été conçus pour une mise en oeuvre sur architecture SIMD. Ils illustrent les besoins de programmation en traitement d'image sur une architecture SIMD à mémoire distribuée [Weems 91]. Pour l'expression et l'implémentation de chaque algorithme, sont détaillés les besoins en termes de structure, de type de données et d'opérateurs associés.

## 2.1 Les structures de données

Comme le montre la Table 1, les données dans ces algorithmes de traitement d'image s'expriment bien sous forme de **tableau**. La structure de données tableau assure une mise en oeuvre claire des images de pixels, des images de labels, des champs de mouvement, etc. La grande majorité des traitements élémentaires et composés peut s'exprimer simplement à l'aide du tableau cartésien à 2 dimensions. Les autres s'appuient sur des tableaux de dimensions supérieures. Par exemple, la labellisation par *Shrinking* demande à pouvoir sélectionner, à l'étape **i** de l'algorithme, le plan défini par **T(*,*,i)**, où le symbole * désigne tous les éléments

[1] Dans la machine $P^3I$, ils sont plûtot traités sur l'unité MIMD.

| Algorithme concerné | Structure de données | Type de données | Opérateurs parallèles |
|---|---|---|---|
| Seuillage | image<br>tableau 2D | de pixels<br>de caractères | de comparaison<br>de sélection |
| Filtrage | image<br>tableau 2D | de pixels<br>d'entiers | arithmétiques<br>de décalage |
| F.F.T. | image<br>tableau 2D | de fréquences<br>de flottants | trigonométriques<br>de permutation |
| Morphologie | image<br>tableau 2D | binaire<br>de booléens | logiques<br>de comparaison |
| Rotation | image<br>tableau 2D | de déplacements<br>d'entiers | de décalage<br>arithmétiques |
| Labellisation par *Shrinking* | image<br>tableau 3D | de réduction bin.<br>de booléens | d'indexation<br>de décalage |
| Labellisation par *Broadcast* | image<br>tableau 2D | de labels<br>entiers non signés | de comparaison<br>de décalage |
| Histogramme | image<br>tableau 2D | de cumuls<br>de vecteurs 1D | de réduction<br>d'indirection |
| Segmentation par *Region growing* | arbre<br>tableau 2D | quaternaire<br>de tableaux 2D | de selection<br>d'indexation |
| Segmentation par *transformé Hough* | image<br>tableau 3D | de paramètres<br>d'entiers | d'indirection<br>de décalage |
| Estimation de mouvement | champs<br>tableau 2D | de mouvement<br>d'entiers signés | de réduction<br>de permutation |

Table 1 : résumé des besoins d'expression en traitement d'images en vue d'une mise en oeuvre sur architecture SIMD

d'une dimension. La transformée de Hough nécessite l'expression de **T(*,*,I)** où **I** est l'image, à un instant donné, des pixels à prendre en compte dans le traitement. Et de ce point de vue, des langages tels que Paralation C [Sabot 88] et Vcode [Blelloch 90], dont la structure de base est le vecteur 1D, s'adaptent mal à la notion d'image multi-dimensionnelle et ne répondent pas bien à l'écriture des algorithmes dans ce domaine.

Le type des données à traiter reste généralement très simple ; il faut juste noter que la notion de pixel ne définit pas un seul type de donnée. Sont regroupées sous cette appellation les données binaires, de type caractère, les entiers signés ou non signés, les couleurs codées sur 24 bits, les valeurs flottantes pour les complexes, etc... La variété des images demande à pouvoir gérer précisemment la dynamique des données afin d'économiser la mémoire.

## 2.2 Les opérateurs

Quant aux opérateurs parallèles, nous pouvons distinguer 2 classes : d'une part les opérateurs locaux (*elementwise*) et d'autre part les opérateurs globaux (*aggregate*) [Blelloch 91]. Il s'agit précisément d'opérateurs parallèles dans un **langage orienté-tableau.**

- Les opérateurs locaux s'appliquent simultanément sur chaque pixel d'une image, sur chaque élément d'un tableau. On note le besoin d'opérateurs arithmétiques, logiques, de comparaison. Les opérateurs locaux peuvent aussi être conditionnels, dépendre d'une condition sur chaque élément : on parle alors de sélection spatiale, de sous-espace actif de tableau. On peut exprimer ces traitements massivement parallèles par une construction itérative du style FORALL [Albert 91] ou PAR() [Bagrodia 90a] qui indique explicitement la nature parallèle des opérations. Mais plus simplement, les traitements peuvent s'exprimer directement par des opérateurs tableau et le type des données permet alors de détecter ce parallélisme massif. Seule l'expression des coordonnées cartésiennes des pixels peut nécessiter une telle construction itérative [Chen 92]. C'est le cas par exemple pour la rotation d'image et l'estimation de mouvement, par exemple.

- Les opérateurs globaux ont un comportement que l'on ne peut pas décrire indépendamment pour chaque pixel. Ils agissent sur la structure même de l'image (indirection, indexation et réduction) ou sur son contenu globalement (décalage, permutation). Nous avons déjà vu en 2.1 l'indirection et l'indexation des tableaux. Quant aux opérateurs de réduction, leur utilisation se limite exclusivement à l'extraction d'un résultat scalaire à partir d'un tableau multi-dimensionnel. C'est le cas, par exemple, de l'estimation de mouvement par *Block matching* où la réduction de l'image de corrélation à son minimum permet de déterminer le mouvement effectué. L'opérateur de permutation, appliqué notamment dans la rotation d'image, consiste à changer les éléments d'un tableau T selon une image conformante (de même forme, de même taille) I d'indices. La complexité de cet opérateur et l'absence de routeur global sur l'unité SIMD nous conduisent à laisser l'utilisateur exprimer et programmer lui-même l'opérateur à l'aide de l'opérateur de décalage régulier. Il existe dans la littérature de nombreux algorithmes qui permettent de l'exprimer ainsi [Reeves 85]. L'opérateur de décalage régulier, quant à lui, constitue une primitive largement sollicitée dans tous les algorithmes abordés. Son utilisation est intensive et impose une expression claire et une implémentation performante.

Ce résumé illustre les besoins des algorithmes de traitement d'image, en termes d'expression et de performance dans un contexte massivement parallèle. Nous avons identifié les besoins en termes de structure de données de base (le tableau multi-dimensionnel) et d'opérateurs associés. Nos conclusions mettent en avant le concept de *langage orienté-tableau*, notion qui n'est pas nouvelle. Elle a déjà fait l'objet de travaux [Sabot 88] [Blelloch 91] [Metcalf 90] et des syntaxes originales ont été proposées.

# 3 Présentation du langage IPF

Le langage IPF (Image Processing Fortran) est basé sur le standard Fortran 90 décrit dans [Metcalf 90]. Notre choix s'est porté sur cette syntaxe parce qu'elle couvre les besoins d'expression tels que nous les avons définis dans la Section 2. Fortran 90 est le langage orienté-tableau de référence et ses opérateurs parallèles assurent une bonne expressivité de nos algorithmes de traitement d'image. Nous avons fait ce choix également parce qu'il existe déjà des outils de compilation qui supportent cette syntaxe, par exemple *Sigma* [Gannon 92]. *Sigma* propose un parseur et des outils de transformation de programmes. La portabilité des applications est, à cet égard, assurée et nos choix de mise en oeuvre destinés à $P^3I$ peuvent être ainsi prototypés et validés sur DECmpp 12000. Des tests comparatifs avec le compilateur de référence DEC HPF 3.1 ont été effectués : les résultats sont détaillés dans la Section 4. Nous exposons maintenant en détail les choix syntaxiques et sémantiques.

## 3.1 Notion de tableau parallèle

Pour la déclaration des tableaux en IPF, nous reprenons la syntaxe du standard Fortran 90. Tous les tableaux sont déclarés de la même manière. Prenons, pour illustrer cela, l'exemple de la convolution d'une image RVB 8 bits I par un filtre F 3x3 d'entiers. L'algorithme consiste, à chaque étape, à décaler l'image I, à appliquer un coefficient du filtre en tout point de I et à cumuler dans l'image résultat R. La partie déclarative, étant donné que l'allocation est statique, est la suivante :

```
integer, parameter      :: haut = 512, larg = 384, prof = 3
character               :: I(haut,larg,prof), R(haut,larg,prof)
integer                 :: F(3,3)
```

Un tableau peut être alloué à la compilation de deux manières. Soit il est placé sur le séquenceur de l'unité SIMD, auquel cas il est qualifié de *tableau scalaire*. Soit il est distribué sur le réseau de processeurs, auquel cas il est qualifié de *tableau parallèle*. L'exemple de la convolution montre que la déclaration en IPF ne permet pas de distinguer les tableaux parallèles des tableaux scalaires. La décision de considérer les images I et R comme des tableaux parallèles et le filtre F comme un tableau scalaire est prise non pas en fonction de leur déclaration mais après analyse de leurs occurences dans le programme. Définissons pour cela les notions d'*occurence parallèle* et d'*occurence scalaire*. Pour les besoins d'expression clairement identifiés dans la Section 2, seules deux formes d'occurences de tableau sont dites parallèles :

- les occurences entières, lorsque le tableau est exprimé par son symbole seul, quel que soit son nombre de dimensions et d'éléments. Par exemple, I est l'occurence entière de l'image I. Cette forme d'occurence est parallèle et permet d'exprimer des calculs locaux sur des tableaux de mêmes tailles,

- les occurences indexées, lorsque le tableau est indexé de manière à exprimer un sous-espace bi-dimensionnel. I(:,:,2) et I(:,:,V) sont les deux seules formes indexées considérées comme des occurences parallèles. V doit être soit une valeur scalaire, soit un tableau déclaré par *integer :: V(haut,larg)*. Nous avons vu, dans la Section 2.1, que ces deux formes correspondaient à un besoin d'expression dans certains algorithmes tels que la labellisation par *shrinking* et la segmentation par *transformée de Hough*.

Toutes les autres formes d'occurences de tableau autorisées en IPF sont qualifiées d'occurences scalaires. Il s'agit des occurences décrites par une valeur scalaire dans chacune des dimensions du tableau. Par exemple, F(1,2) est une occurence scalaire valide du filtre F. F(i,j) en est également une si et seulement si i et j ont été déclarés au préalable par *integer :: i,j*. Ainsi un tableau est dit parallèle si et seulement si toutes ses occurences dans le programme sont parallèles. Sinon il est dit scalaire. Le compilateur détermine ainsi automatiquement la nature des tableaux en IPF et leur allocation.

Quant au type des éléments, le langage supporte actuellement les booléens (*LOGICAL*), les

entiers (*CHARACTER, INTEGER*) et les flottants (*REAL*). L'extension aux types structurés peut présenter certes un intérêt du point de vue de l'expressivité des programmes mais elle n'a pas été mise en oeuvre. Nous avons vu (Section 2.1) que la notion de pixel ne correspond pas forcément à un seul type de donnée ; un pixel peut être codé sur 1, 8, 12, 24, 32 bits voire plus. Le standard Fortran 90, à cet égard, dispose d'un mécanisme (*<type>*<nb_bits>*) qui permet de définir des sous-types, de n'allouer que le minimum de mémoire sur le réseau SIMD. Les processeurs élémentaires disposent de peu de mémoire interne et le mécanisme de définition de sous-types permet de gérer au mieux cette ressource.

## 3.2 Le rangement des tableaux

Nous avons vu dans la Section 2.2 que les algorithmes de traitement d'image sur architecture SIMD requièrent globalement d'une part des calculs locaux et d'autre part des décalages réguliers d'image. En effet, l'essentiel des traitements se fait sur le voisinage géographique de chaque pixel. Cela est bien particulier au domaine et, à cet égard dans un contexte massivement parallèle, le rangement des tableaux sur la grille de processeurs élémentaires est fondamental. Il nous faut trouver un placement des données tel que les performances soient les meilleures. La première idée est de répartir équitablement les éléments dans la mémoire des processeurs, pour obtenir un maximum de parallélisme lors des opérations locales. La seconde idée est de favoriser l'accès au voisinage géographique, pour minimiser les communications entre processeurs, lors des décalages d'images.

Nous avons comparé pour cela les placements en **mode bloc** et en **mode feuille**, qui constituent les deux modèles d'allocation des données les plus utilisés sur architecture SIMD à mémoire distribuée. Le Figure 2 présente le placement en **mode bloc**. Le placement en mode feuille, utilisé notamment par le compilateur DEC HPF 3.1, consiste également à répartir équitablement les données sur les processeurs mais de telle manière que deux pixels voisins dans l'image se situent toujours sur deux processeurs voisins. Le décalage d'image est donc plus coûteux avec le mode feuille qu'avec le mode bloc.

Le mode feuille est intéressant en calcul numérique par exemple, pour la gestion du *load balancing*. Mais le placement en mode bloc est ici le meilleur compromis, compte tenu de la nature et de la fréquence des opérations parallèles. D'un coté, chaque processeur se voit alloué le même nombre d'éléments : potentiellement, le parallélisme est maximal. Et de l'autre coté, la localité est favorisée : deux pixels contigüs dans l'image sont soit localisés sur le même processeur, ce qui est le cas général, soit situés en frontière de sous-bloc et placés alors sur deux processeurs voisins.

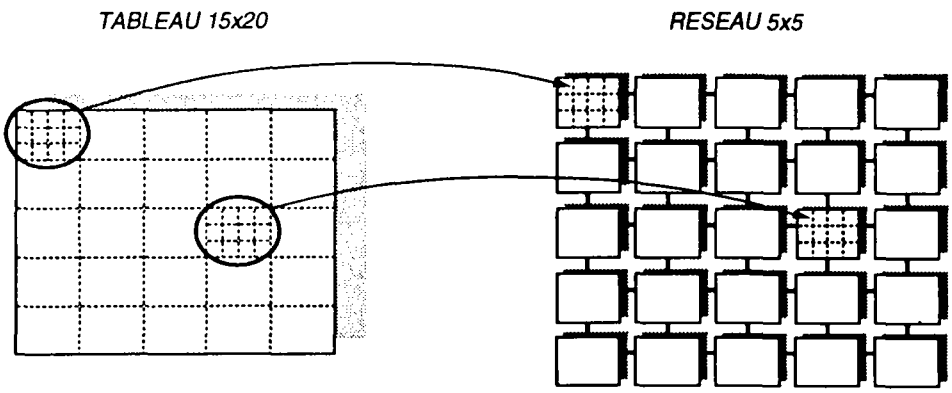


Figure 2 : découpage en mode bloc d'un tableau à deux dimensions.

## 3.3 Les opérateurs locaux

Un opérateur local (*elementwise operator*) s'applique simultanément à tous les éléments d'un tableau, de façon identique. En IPF comme en Fortran 90, la sémantique des opérateurs scalaires a été étendue de manière à pouvoir les utiliser sur les tableaux. Nous trouvons les opérateurs arithmétiques (+, -, *, /, *ABS*, ...), logiques (*.NOT.*, *.AND.*, *.OR.*, ...), de comparaison (<, >, ==, *etc*) et également les fonctions trigonométriques. Sur l'algorithme de convolution précédent, le produit d'un coefficient du filtre par l'image I accumulé dans le tableau résultat s'exprime, en IPF, par l'instruction :

$$R = R + (I * F(i,j))$$

R (respectivement I) est une occurence parallèle du tableau R (respectivement I), F(i,j) est une occurence scalaire du tableau F, l'instruction est dite implicitement parallèle. R et I doivent être des tableaux *conformants*, c'est-à-dire ayant le même nombre de dimensions et le même nombre d'éléments dans chaque dimension. Cette contrainte assure une sémantique correcte de l'instruction. Nous avons vu, en 2.2, que les opérateurs locaux peuvent être conditionnels, c'est-à-dire s'appliquer seulement aux éléments de tableau qui vérifient une condition. Nous disposons, pour exprimer cette sélection d'éléments, d'une construction (*WHERE <condition> <body1> [ELSEWHERE <body2>] ENDWHERE*) appelée conditionnelle spatiale.

```
Where (I >= seuil_min)
        R = I / 2
Elsewhere
        R = 0
Endwhere
```

Elle peut être imbriquée en IPF. La condition est toujours une expression parallèle et les occurences parallèles présentes dans les alternatives du WHERE sont conformantes.

Dans la Section 2.2, nous avons également soulevé le besoin d'exprimer les coordonnées cartésiennes de tableau, en traitement d'image. Le standard Fortran 90 propose une solution avec le constructeur générique de boucles parallèles FORALL. Nous lui préférons, en IPF, une fonction intrinsèque originale, *COORD(<tableau T>,<dimension D>)*. Elle rend un tableau conformant à T, dont les éléments sont les coordonnées dans la dimension D précisée. Elle apparait exclusivement dans les expressions parallèles. Nous pouvons ainsi exprimer, plus simplement qu'avec le FORALL, sans expliciter d'indice de boucles, tous les calculs qui dépendent de la géographie des pixels, ce qui se traduit par de meilleures performances.

L'expression des sections de tableaux, de la forme T(:,i) par exemple, n'est pas autorisée en IPF comme en Fortran 90 mais l'utilisation combinée de la conditionnelle spatiale WHERE et de la primitive COORD(...) permet de pallier à ce manque sans perte d'efficacité. On exprime simplement T(:,i) par :

```
Where (COORD(T,2)==i)
        ... T ...
Endwhere
```

Ces choix présentent l'avantage de ne laisser au programmeur qu'une seule manière simple d'exprimer les calculs locaux sur les tableaux. Sans perte d'expressivité, le compilateur peut détecter ainsi systématiquement les sections parallèles de programme.

## 3.4 Les opérateurs globaux

IPF est un langage orienté-tableau où les opérateurs globaux s'expriment sous forme de fonctions intrinsèques telles qu'elles sont définies en Fortran 90. Définies avec des paramètres variables, en nombre et en type, les fonctions intrinsèques sont intégrées, analysées et expansées lors de la compilation. Elles présentent l'avantage d'éviter les séquences d'appel souvent coûteuses, contrairement aux fonctions en librairie. IPF possède les fonctions intrinsèques suivantes :

- Les fonctions de réduction de tableau selon un opérateur commutatif et associatif. Nous avons retenu *ALL* et *ANY* pour les tableaux de booléens, *SUM*, *COUNT*, *MAXVAL* et *MINVAL* pour les tableaux numériques. Elles respectent toutes la même syntaxe : *<fonction> (<tableau> [,<condition>])*. Le résultat est systématiquement un scalaire, ce qui signifie qu'un appel à une fonction de réduction peut apparaitre dans n'importe quelle expression, parallèle ou scalaire. Le premier paramètre doit être une occurence parallèle de tableau. La condition, elle, est optionnelle et permet de n'effectuer la réduction que pour un sous-ensemble d'éléments. Cela est utile, par exemple, pour calculer la surface d'un objet :

  *surface = COUNT(I,(I==label_objet))*

- La fonction de décalage régulier de tableau, suivant une dimension. La syntaxe Fortran 90 *(CSHIFT(<tableau>,<distance>,<dimension>)* autorise des décalages d'une distance, négative ou positive, quelconque. Notons qu'il y a rebouclage des données aux extrémités du tableau. Le résultat de cette primitive intrinsèque est un tableau décalé, conformant au tableau passé en paramètre. Dans la version initiale de IPF, seul le décalage d'un même tableau (I = CSHIFT(I,dist,dim)) est permis. Cette limitation est dûe au schéma de compilation particulier que nous avons développé. Les raisons en sont exposées dans la Section 4.2.

- Les fonctions d'entrée/sortie. Propres à IPF et dédiées à la manipulation d'images, elles ont été imaginées afin de faciliter la mise au point des programmes. Nous disposons de trois primitives : *LOAD(<fichier>,<tableau>)* qui charge une image dans un tableau, *STORE(<tableau>,<fichier>)* qui effectue l'opération inverse,

*DISPLAY(<tableau>)* qui permet de visualiser le contenu d'un tableau.

## 3.5 Le séquencement

Du point de vue du séquencement des programmes, les besoins d'expression sont relativement simples. Outre la conditionnelle classique *IF <condition> THEN <body1> [ELSE <body2>] ENDIF*, les constructeurs de boucle *DO <compteur> <body> ENDDO* et *WHILE <condition> <body> ENDWHILE* assurent le contrôle des traitement itératifs.

IPF n'est clairement pas à usage général comme l'est le standard Fortran 90. Nous l'avons conçu pour les besoins d'expression spécifiques du traitement d'image, dans l'optique d'une implémentation performante sur architecture SIMD à mémoire distribuée. En particulier, nous avons soigné l'expression et la manipulation des tableaux de manière à pouvoir détecter automatiquement le parallélisme et à avoir un schéma de compilation très efficace.

La Section suivante décrit les techniques de compilation mise en oeuvre pour IPF et présente les premiers résultats comparatifs de notre prototype avec le compilateur DEC HPF version 3.1.

# 4 Le prototype

Le prototype de compilateur que nous avons réalisé a été conçu pour tester et valider les techniques et schémas de compilation que nous avons choisis. Nous nous sommes concentrés en particulier sur les problèmes de virtualisation des tableaux [Christy 91] [Weiss 91] et de réduction des communications [Knobe 88] [Knobe 90]. Pour faciliter le développement de ce prototype et compte-tenu que IPF reprend la syntaxe de Fortran 90, nous avons utilisé *Sigma* [Gannon 92] qui réalise l'analyse syntaxique et transforme les programmes sources sous forme de graphe. A partir de cette représentation, nous avons développé nos schémas à l'aide des outils de transformation de programmes de *Sigma* ; cela constitue l'essentiel du travail. Enfin, nous avons écrit un simple générateur de code MPL C. Ce générateur sera, à terme, remplacé par un générateur d'assembleur $P^3I$ (en cours de test). La Figure 3 présente la chaine

de compilation du prototype et celle de DEC HPF.

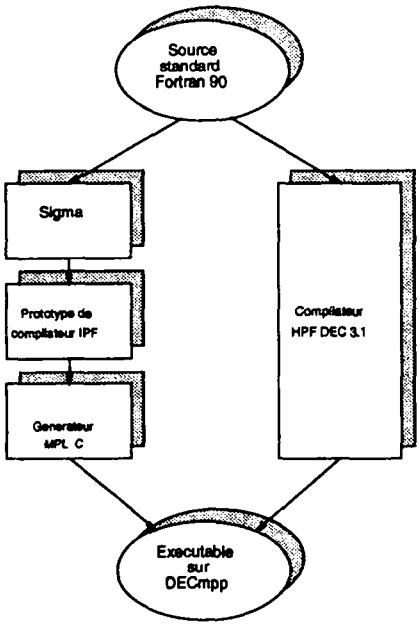


Figure 3 : chaines de compilation du prototype IPF et de DEC HPF 3.1.

Nous avons d'un coté le compilateur DEC HPF qui utilise le placement des données en mode feuille, qui optimise le code produit et gère efficacement l'allocation des registres. DEC HPF génère directement de l'assembleur pour DECmpp 12000. De l'autre coté, le prototype de compilateur IPF exploite le placement en mode bloc mais ne contient aucun module d'optimisation. En particulier à la génération d'assembleur laissée à la charge de MPL C, aucune optimisation de l'utilisation des registres n'est effectuée.

Nous présentons maintenant le détail des travaux, à savoir la technique d'allocation des tableaux, les schémas de compilation appliqués aux fonctions intrinsèques (décalage, réduction, entrée/sortie et calcul de coordonnés) et la méthode de virtualisation appliquée aux opérations locales.

## 4.1 L'allocation des tableaux

Nous avons défini, dans la Section 3.1, la notion de nature de tableau : un tableau en IPF, suivant son utilisation, est scalaire ou parallèle. S'il est scalaire, il est alloué sur l'ACU (Array Control Unit) de l'unité SIMD. S'il est parallèle, il est alloué sur le réseau lequel est par hypothèse une grille torique à deux dimensions (le nombre de processeurs dans chacune des dimensions est passé en paramètre dans la commande de compilation). Nous avons établi un schéma pour ces deux natures de tableau.

- Les tableaux scalaires sont alloués sur l'ACU sous forme de vecteurs mono-dimensionnels. Leur déclaration est modifiée simplement en réduisant le nombre de dimensions à un et en faisant le produit des éléments dans chaque dimension. Nous adoptons le rangement *Rowwise*. La Figure 4 illustre cette transformation pour le filtre F dans l'exemple de la convolution.

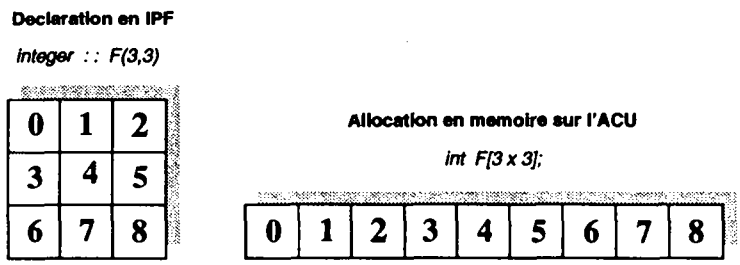


Figure 4 : rangement *Rowwise* des tableaux 2D en IPF

- Les tableaux parallèles sont alloués sur le réseau 2D de processeurs. La taille de la grille est définie par les deux paramètres *nyproc* (nombre de processeurs en vertical) et *nxproc* (nombre de processeurs en horizontal). Nous avons montré dans la Section 3.2 que le placement en mode bloc est le mieux adapté aux algorithmes de traitement d'image et en particulier que la répartition équilibrée des éléments de tableau donne un parallélisme maximum. Nous définissons donc le *ratio de virtualisation* [Christy 91] comme étant le nombre d'éléments alloués sur chaque processeur élémentaire. Ce ratio est calculé d'après la déclaration en IPF et dépend des paramètres du réseau. Pour l'image I de l'exemple, la transformation opérée sur sa déclaration donne, traduit en MPL C, :

*plural char I [((haut+nyproc-1)/nyproc)*((larg+nxproc-1)/nxproc)*def];*

Le ratio de virtualisation est égal au produit des ratios dans chaque dimension. Notons que l'expression *(haut+nyproc-1)/nyproc* tient compte du cas où la constante *haut* n'est pas un multiple de *nyproc* (même remarque pour *larg* et *nxproc*). Le rangement des éléments est le même que pour les tableaux scalaires, à savoir *Rowwise*.

De plus, à chaque tableau parallèle, quelle que soit sa déclaration, on associe trois attributs dont 2 variables scalaires (un indice et une base) et un tableau parallèle auxiliaire. Ces 3 attributs interviennent dans les schémas de transformation des programmes IPF. L'indice est utilisé pour la boucle de virtualisation des instructions parallèles dans lesquelles apparaissent des occurences du tableau. Le besoin d'une base est lié à la technique de compilation appliquée à l'opérateur de décalage régulier CSHIFT. Enfin le tableau parallèle auxiliaire, dont la taille (nyproc,nxproc) est celle du réseau, est utilisé dans le schéma de transformation des fonctions intrinsèques de décalage et de réduction que nous présentons maintenant.

## 4.2 Le décalage régulier : CSHIFT(I,dist,dim)

Le décalage de tableau suivant une dimension est une opération très courante en traitement d'image. Nous l'avons vu pour la permutation des données, pour les calculs sur le voisinage, etc. Cet opération est coûteuse sur une architecture SIMD à mémoire distribuée, en termes de communications inter-processeurs et de mouvement de données intra-processeur car elle doit respecter la virtualisation. C'est pourquoi elle fait l'objet d'un soin tout particulier. Compte-tenu du placement des tableaux parallèles en mode bloc en IPF, l'opération de décalage, du point de vue d'un processeur élémentaire, peut classiquement se décomposer (Figure 5) en deux phases : une de transfert de la bordure au PE voisin et une de réalignement interne des éléments restants.

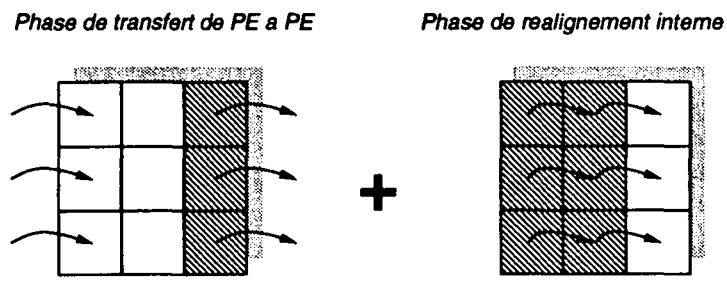


Figure 5 : décomposition en deux phases du décalage d'un tableau 2D placé en mode bloc sur un réseau SIMD à mémoire distribuée

La phase de communication assure la localité des éléments sur les processeurs et la phase suivante de réalignement réordonne en mémoire ces éléments pour conserver l'invariant sur l'ordre de stockage *Rowwise*. Si l'on considère un tableau dont le ratio de virtualisation est $n^2$, le décalage d'une distance de 1 dans une direction quelconque se traduit, pour chaque PE, par :

- O(n) opérations de communication inter-PEs,
- $O(n^2)$ opérations mémoire intra-PEs.

Nous avons développé un mécanisme qui permet de réduire la fonction de décalage aux simples opérations de communication : il s'agit de *l'adressage modulo*. La technique de l'adressage modulo évite tout réalignement interne des données ; elle économise donc $O(n^2)$ opérations mémoire dont le coût sur ce type de machine est proche de celui d'une opération de communication ! Le principe est de considérer le vecteur d'allocation 1D sur chaque PE comme circulaire. Le dernier élément précède le premier élément du vecteur, selon la fonction d'adressage modulo. A partir de là, il existe un schéma de communications entre proceseurs qui conserve l'invariant sur l'ordre de stockage, modulo le ratio de virtualisation. La Figure 6 illustre cette technique. L'origine logique du sous-bloc initialement est à 0, sur tous les processeurs élémentaires.

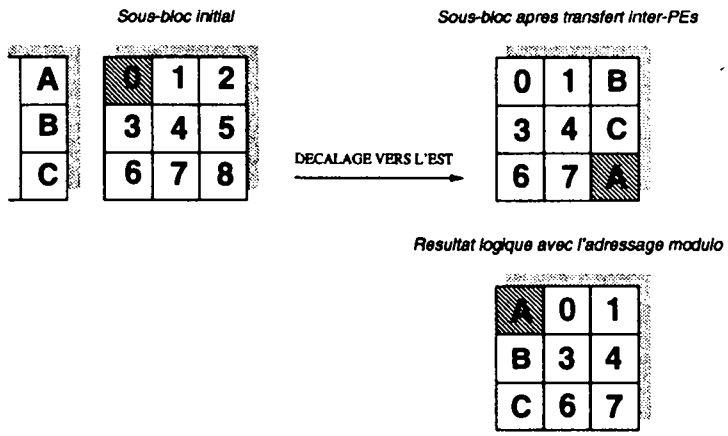


Figure 6 : décalage horizontal vers l'Est d'un tableau 2D (nyproc*3,nxproc*3), selon l'adressage modulo. L'élément grisé constitue l'origine logique du sous-bloc

Sur cet exemple, les éléments A, B et C sont reçus du processeur voisin de gauche et sont

stockés à la place de ceux envoyés au voisin de droite, selon une permutation particulière. L'origine logique des sous-blocs est alors modifiée (A devient la nouvelle origine) et l'invariant est toujours vérifié, modulo le ratio de virtualisation.

De façon générale donc, nous avons besoin d'associer à chaque tableau parallèle une information supplémentaire : son origine logique (ou base). Cette base, la même pour tous les processeurs, repère le début logique des sous-blocs du tableau. C'est une information scalaire mise à jour à chaque décalage régulier. Le schéma général de transformation, traduit en MPL C, pour une instruction du type T = CSHIFT(T, dist, dim) est le suivant :

```
for (d=0;d<abs(dist);d++)
 {
        ratio = ratio(d1)*...*ratio(dn);
        pas_int = 1;
        pas_ext = dist/abs(dist) * ratio(ddim)*...*ratio(dn);
        if (dist>0) bas_T = (bas_T - (pas_ext/ratio(ddim))) mod ratio;
        ind_T = bas_T;

        for (i=1;i<=(ratio(ddim+1)*...*ratio(dn));i++)
          {
            aux_T = T[ind_T];
            for (j=2;j<=(ratio(d1)*...*ratio(ddim-1)));j++)
              {
                  xnet(f(dist,dim))[1].T[ind_T] = T[(ind_T + pas_ext) mod ratio];
                  ind_T = (ind_T + pas_ext) mod ratio;
              }
            xnet(f(dist,dim))[1].T[ind_T] = aux_T;
            ind_T = (ind_T + pas_ext + pas_int) mod ratio;
          }
        if (dist<0) bas_T = (bas_T - (pas_ext/ratio(ddim))) mod ratio;
 }
```

Nous avons évalué ce schéma de compilation par rapport au schéma classique composé de deux phases et par rapport à celui développé par le compilateur DEC HPF 3.1 disponible sur la DECmpp 12000. Les Figures 7 et 8 présentent les résultats d'un décalage d'une position d'un tableau à deux dimensions ($n^2$) d'entiers. Sur toutes les courbes qui suivent, la taille indiquée en pixels correspond à la variable n des tableaux. Les temps obtenus sont eux proportionnels à $n^2$, le nombre d'éléments des tableaux. Cela explique pourquoi la forme générale des courbes n'est pas linéaire. Nous avons choisi le type entier pour pouvoir comparer nos résultats à ceux de DEC HPF qui n'autorise pas les tableaux parallèles de

caractères.

La Figure 7 fait apparaitre un phénomène d'escalier que l'on retrouve dans toutes les courbes suivantes. Il s'agit, pour IPF, d'une question d'allocation mémoire : il existe une marche entre le résultat d'une image $1024^2$ et une image $1025^2$. Dans le premier cas, $8^2$ éléments sont alloués sur chaque PE, dans l'autre $9^2$. Pour DEC HPF, le phénomène constaté est que lorsque la taille de l'image n'est pas un multiple du nombre de processeurs, le schéma de compilation de DEC HPF est lourd : il faut deux fois plus de temps pour décaler une image $1088^2$ que pour décaler une image $1024^2$.

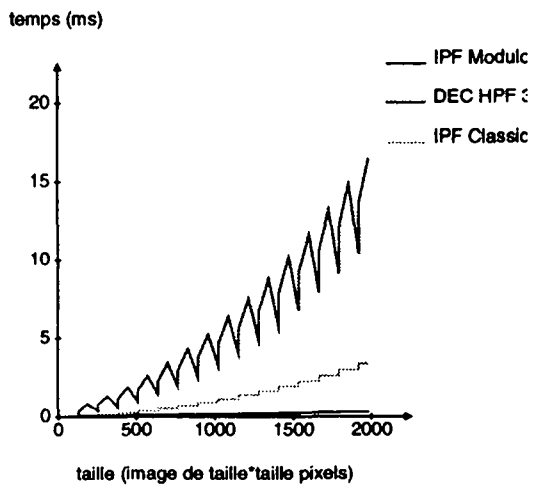


Figure 7 : résultat du décalage d'un tableau 2D d'entiers selon l'axe horizontal

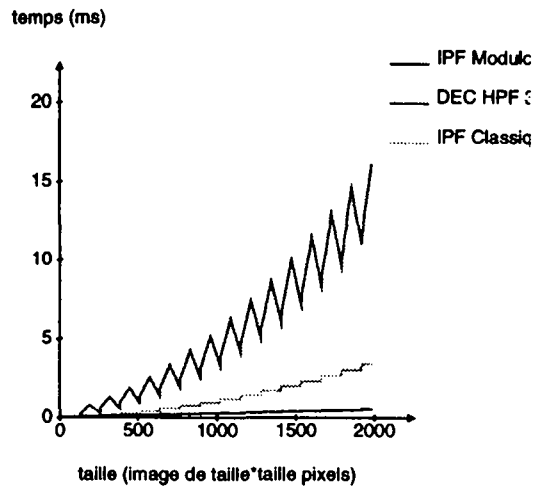


Figure 8 : résultat du décalage d'un tableau 2D d'entiers selon l'axe vertical

Quant aux résultats proprement dits, la Figure 7 montre que la version modulo est plus performante que la version classique (rapport de 2.8 pour une image $512^2$). Cela s'explique par la technique d'adressage modulo. La version classique du prototype IPF est elle-même plus efficace que le schéma développé par DEC HPF (rapport de 3.6 pour une image $512^2$). La différence s'explique cette fois-ci par le rangement en mode bloc que nous avons adopté et qui correspond bien aux besoins ici.

La Figure 8 donne les résultats pour le décalage d'image vertical. Les rapports obtenus sont sensiblement les mêmes que ceux du décalage horizontal. Les résultats pour les dimensions supérieures ne sont pas donnés dans la mesure où ils confirment simplement les enseignements tirés des deux figures précédentes. Cette technique d'adressage modulo permet donc d'accélérer les opérations de décalage. Par contre, elle génère un coût supplémentaire pour les opérations locales, dans la mesure où l'operateur modulo est implémenté en logiciel. Nous évaluons ce coût dans la Section 4.6.

## 4.3 Les réductions

Les primitives de réduction suivent toutes le même schéma de traduction. Elles acceptent un ou deux paramètres, le tableau sur lequel s'effectue la réduction et éventuellement un masque. Etant donné que le résultat est une valeur scalaire, on peut trouver une primitive de réduction dans toute expression, qu'elle soit scalaire ou parallèle.

Le schéma se décompose en une boucle de fusion locale et en un arbre de réduction logarithmique symbolisé par la fonction MPL reduceXXX(). Voici l'exemple de la traduction de l'instruction b = ANY(T).

```
ratio = ...
aux_T = false;
for (ind_T=0; ind_T<ratio; ind_T++)
  { aux_T = aux_T || T[ind_T]; }
b = reduceOr8(aux_T);
```

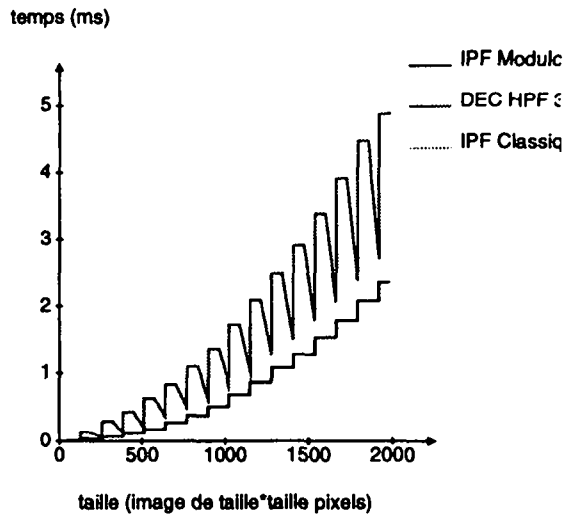


Figure 9 : résultat de la primitive de réduction ANY() pour un tableau 2D d'entiers

La Figure 9 ne fait apparaitre que deux courbes distinctes. Le schéma de traduction d'une

opération de réduction est le même pour les techniques d'adressage classique et modulo ; les deux courbes sont confondues. Le temps en DEC HPF est légèrement supérieur à celui de IPF. Cela est dû probablement à la mise en oeuvre plus efficace de l'arbre de réduction logarithmique *reduceOr8(...)* en MPL C qu'en DEC HPF.

## 4.4 Les entrées/sorties

Cette fonction charge le fichier F dans le tableau T réparti sur le réseau de processeurs. La primitive MPL **mpiread** considère le caractère (8 bits) comme type de base. Il faut donc prendre en compte non seulement le nombre d'éléments mais aussi leur type. Il faut que l'égalité taille(F) = Produit(dimensions(T))*taille(type(T)) soit respectée. Si un tableau T est déclaré par *integer :: T(512,512)* alors le fichier F doit contenir exactement 512*512*4 caractères (ou octets).

Lors de la phase de génération de code, on génère un bloc local contenant une déclaration de tableau temporaire dont à besoin MPL lors de l'appel à la primitive d'entrée/sortie **mpiread2d8**. Ce tableau temporaire est déclaré comme le tableau T. L'initialisation d'un tableau par l'appel à LOAD entraîne la remise à 0 de sa base (bas_T).

```
{
  plural int work[16];
  mpiread2d8("toto.ima", 4*4, 4, T, work);
  bas_T = 0;
}
```

La primitive STORE sauvegarde le tableau T réparti sur le réseau de processeurs selon un mapping M(T) dans le fichier F. Le schéma de traduction en MPL est sensiblement le même, si ce n'est qu'une phase intermédiaire de recopie est nécessaire.

On génère un bloc local contenant une déclaration de tableau temporaire dont à besoin MPL lors de l'appel à la primitive d'entrée/sortie **mpiwrite2d8**. Ce tableau temporaire est déclaré de la même manière que le tableau T. Se pose un problème concernant la base du tableau T : elle est à priori différente de zéro et donc un réalignement (changement de base) est nécessaire avant la sauvegarde... ce qui demande à réinitialiser bas_T à 0 finalement.

```
{
 plural int work[16];
 int i, ratio;
 ratio = 16;
 for (i=0; i<ratio; i++)
  { work[i] = T[(bas_T+i)%ratio] }
 for (i=0; i<ratio; i++)
  { T[i] = work[i]; }
 mpiwrite2d8(T, 4*4, 4, "titi.ima", work);
 bas_T = 0;
}
```

## 4.5 La primitive COORD(T,D)

La fonction COORD rend en résultat un tableau conformant au tableau en paramètre T et dont les éléments sont les coordonnées cartésiennes dans la dimension D indiquée. Cette primitive, de part la nature du résultat qu'elle rend, se situe toujours dans une expression parallèle, soit de calcul, soit de comparaison. Toute occurrence de COORD(T,D) se traduit donc simplement (Table 2) par une expression fonction de **ind_T**, l'indice de la boucle de virtualisation englobante, de **bas_T**, la base du tableau, de **ixproc** et **iyproc**, les coordonnées de chaque processeur.

| dimension | Expression associée |
|---|---|
| 1 | (iyproc*ratio($d_1$))+(((ind_T-bas_T)%ratio)/(ratio/ratio($d_1$))) |
| 2 | (ixproc*ratio($d_2$))+(((ind_T-bas_T)%(ratio/ratio($d_1$))/(ratio/(ratio($d_1$)*ratio($d_2$)))) |
| i (i>2) | ((ind_T-bas_T)%(ratio/(ratio($d_1$)*...*ratio($d_{i-1}$)))/(ratio/(ratio($d_1$)*...*ratio($d_i$)))) |

Table 2 : formule de calcul des coordonnées en fonction de la dimension

Un tel schéma est intéressant dans la mesure où il ne nécessite aucun stockage intermédiaire. Pas besoin de déclarer un tableau pour recueillir les coordonnées. Il n'existe pas de telle fonction intrinsèque en DEC HPF. Si nous voulons l'exprimer en DEC HPF sous forme de fonction, il faut allouer et calculer effectivement un tableau de coordonnées par une boucle Forall.

## 4.6 Les opérations locales

Les opérations élémentaires appliquées aux tableaux sont toujours situées dans une instruction d'affectation parallèle, qui comporte en partie droite une expression où les occurrences de tableaux sont conformantes et alignées. Le même schéma de génération est appliqué pour les instructions de conditionnelle spatiale WHERE. L'imbrication de plusieurs WHERE permet l'expression de conditions plus complexes et ne modifie pas le schéma de compilation. Seule l'opération de décalage régulier CSHIFT ne peut figurer dans une expression parallèle. Cette restriction est dûe à l'adressage modulo et liée au fait que l'espace d'itération n'est pas le même dans les deux cas.

Etant donnée la représentation interne des tableaux (vecteur 1D alloué statiquement et base associée bas_T), de telles opérations se traduisent par une boucle de virtualisation où chaque indice parcourt l'espace en partant de la base du tableau modulo le ratio de virtualisation. Quel que soit le nombre de dimensions des tableaux, le schéma est toujours applicable sous réserve que toutes les occurences parallèles de l'instruction soient comformantes et alignées. Par exemple, l'extrait de programme IPF :

```
character :: Ima1(512,512), Ima2(512,512)
...
Ima1 = 2 * Ima1 - Ima2
```

se traduit en MPL C par

```
ratio = 16;
ind_Ima1 = bas_Ima1;
ind_Ima2 = bas_Ima2;
for (i=1; i<=ratio; i=i+1)
  {
    Ima1[ind_Ima1] = 2 * Ima1[ind_Ima1] - Ima2[ind_Ima2];
    ind_Ima1 = (ind_Ima1 + 1) % ratio;
    ind_Ima2 = (ind_Ima2 + 1) % ratio;
  }
```

Les Figures 10 et 11 présentent les performances comparées de IPF et de DEC HPF version 3.1 concernant les opérations locales. Deux sortes d'instruction ont été évaluées. La première correspond à une instruction d'affectation simple (Ima1 = Ima2), la seconde représente des calculs arithmétiques mettant en oeuvre deux images, une variable scalaire et des valeurs constantes.

La version classique du prototype de IPF est légèrement meilleure que la version modulo : la différence entre les deux, plus sensible sur la Figure 11, est simplement le temps passé à gérer l'opérateur modulo. On constate un rapport important en faveur de DEC HPF, de manière générale. Ce rapport est logiquement croissant avec la taille des images. Il s'explique à deux niveaux. D'abord au niveau du prototype de IPF qui n'effectue aucune optimisation sur le code qu'il transforme (les boucles inutiles ne sont pas éliminées par exemple). Ensuite au niveau du compilateur MPL C dont le rôle se limite à générer de l'assembleur pour DECmpp (pas d'allocation de registres en particulier). Pour sa part, DEC HPF génère directement de l'assembleur et sait optimiser, aussi bien sur le code produit que sur la gestion mémoire.

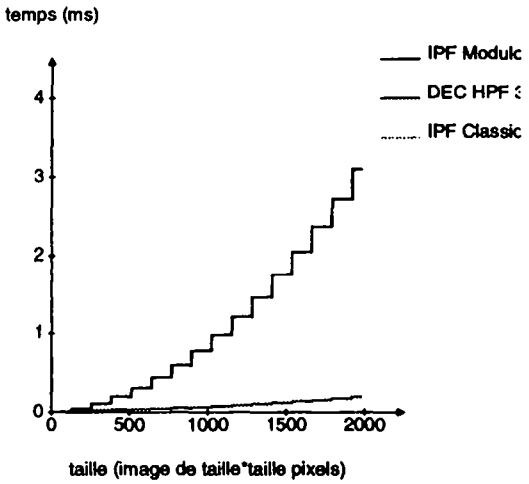


Figure 10 : résultat de l'affectation entre deux tableau 2D d'entiers (Ima1 = Ima2)

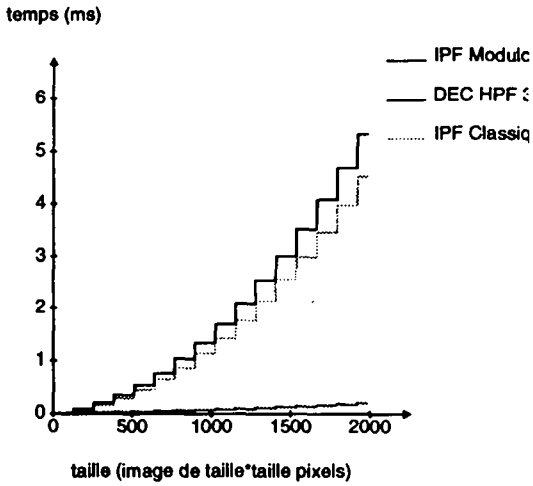


Figure 11 : résultat du calcul local (Ima1 = 2*Ima2 + val/3)

Ces premiers résultats donnent une idée des performances du compilateur. L'évaluation des opérations de base du langage donne un aperçu de la qualité des schémas de compilation mis en oeuvre pour IPF. Il est néanmoins difficile de tirer des conclusions sur la technique de l'adressage modulo, de savoir dans quelle mesure elle est meilleure que la solution classique. L'expérimentation du prototype sur de véritables algorithmes de traitement d'image est un élément indispensable pour une meilleure analyse des performances. C'est ce que nous

proposons dans la section suivante.

# 5 Expérimentation

Sont présentés ci-après les résultats comparatifs du prototype de compilateur IPF et de DEC HPF sur quelques algorithmes de traitement d'image de bas et moyen niveau. Nous avons retenu un échantillon représentatif des besoins d'expression :

- la segmentation par transformée de Hough [Kannan 89],
- la détection de contour par le calcul de gradient (Laplace avec un filtre 3x3, Nevatia avec 6 filtres 5x5) [Rosenfeld 82] [Fang 89],
- la rotation d'image selon une angle quelconque [Strong 82],
- la labellisation d'objets (Broadcast, Shrinking) [Hsu 90] [Hambrusch 89] [Cypher 90],
- l'estimation de mouvement en chaque point (Différentielle, Block matching) [Williams 86] [Heitz 90].

## 5.1 Transformée de Hough

La transformée de Hough, limitée à deux paramètres, permet par exemple de détecter et d'extraire d'une image de points, des droites. L'algorithme que l'on se propose d'implanter [Kannan 89] pose des problèmes en DEC HPF 3.1. Il s'agit de faire un histogramme *Hist* en fonction du couple de paramètres $(r,\theta)$, r étant le rayon et $\theta$ l'angle. A la fin de la phase horizontale, le vecteur monodimensionnel H(haut,i,:) contient la ième ligne de l'espace des paramètres correspondant à $\theta_i$. A titre d'illustration, on trouvera un extrait du programme source écrit en IPF.

```
!---- déclaration des constantes ----

integer, parameter :: haut = 256, larg = 256          ! taille des tableaux
integer, parameter :: def = 256                       ! résolution en r
real, parameter :: pi = 3.14159

!---- déclaration des tableaux ----

integer :: Src(haut,larg)                             ! image source
integer :: Som(haut,larg)                             ! somme intermédiaire
real :: Theta(haut,larg)                              ! image des angles θi
integer :: Rc(haut,larg), R(haut,larg)                ! image des rayons
integer :: Hist(haut,larg,def)                        ! histogramme
integer :: i

!---- initialisations ----

...
Theta = 2 * COORD(Src,2) * pi / larg                  ! initialisation des θi
Rc = def * fp_cos(Theta) / larg                       ! calcul du pas d'incrément

!---- phase de cumul vertical ----

R = (def * (COORD(Src,1)*fp_cos(Theta) + COORD(Src,2)*fp_sin(Theta)) / larg
DO i=1,haut-1
  WHERE (COORD(Src,2)==i)                             ! au bout de i itérations
      R = (def * COORD(Src,2) * fp_sin(Theta) + 1) / larg   ! mise à jour de R
  ENDWHERE
  WHERE (Src==1)                                      ! pour chaque point allumé
      Hist(:,:,R) = Hist(:,:,R) + 1                   ! cumul dans le R calculé
  ENDWHERE
  Src = CSHIFT(Src,1,1)                               ! décalage circulaire de
                                                      ! Src vers le bas
  R = R + Rc                                          ! incrément de R
ENDDO

!---- phase de cumul horizontal ----

Som = 0                                               ! initialisation du cumul
R = 1                                                 ! initialisation du rayon
DO i=1,2*(larg-1)
  WHERE ((i>COORD(Src,2)) .AND. (i<(COORD(Src,2)+larg)))   ! pour les colonnes actives
      Som = Som + Hist(:,:,R)                         ! cumul partiel horizontal
      Hist(:,:,R) = Som                               ! stockage dans l'histogramme
      R = R + 1                                       ! passage au rayon suivant
```

```
  ENDWHERE
  Som = CSHIFT(Som,1,2)                    ! décalage des cumuls à droite
  WHERE (COORD(Som,1)==0)                  ! en première colonne (gauche)
       Som = 0                             ! raz du cumul de la ligne
  ENDWHERE
ENDDO
```

Le compilateur DEC HPF refuse les expressions d'indirections du style *Hist*(:,:,R) avec R tableau à deux dimensions de même taille que les deux premières dimensions de *Hist*. Or la sémantique est claire : il s'agit d'adresser pour chaque pixel (i,j) la case *Hist*(i,j,R(i,j)).

En IPF, cette écriture est correctement traitée et exploite en l'occurence les possibilités d'adressage indirect dont disposent les processeurs élémentaires de toutes les machines SIMD actuelles. La Table 3 présente donc seulement les performances de IPF pour la transformée de Hough dans l'espace à deux dimensions. Les résultats de la version modulo sont très légèrement inférieurs à ceux de la version classique. Cela signifie que les opérations locales, plus coûteuses avec l'adressage modulo, prennent le pas sur les opérations de décalage dans cet algorithme.

| Taille de l'image de référence | Transformée de Hough (en s) | | |
|---|---|---|---|
| | IPF 1.0 | IPF vc | HPF 3.1 |
| (128,128) | 0,207 | 0,187 | - |
| (129,129) | 1,285 | 1,152 | - |
| (256,256) | 1,285 | 1,152 | - |
| (384,384) | 4,031 | 3,607 | - |

Table 3 : évaluation de la transformée de Hough

## 5.2 Détection de contour

La détection de contour sur une image par la technique du gradient peut se résumer au produit d'une image par un ou plusieurs masques. Nous présentons ici les résultats de deux algorithmes de détection de contour par cette technique. Le filtre de Laplace se résume à appliquer un masque 3x3 et la méthode de Nevatia s'appuie sur l'enchainement de 6 filtres

5x5 pour lesquels on ne conserve que le maximum local. Quel que soit le contenu de l'image traitée et les valeurs de masque, le temps de calcul est constant.

Les Figures 12 et 13 montrent très clairement que IPF est plus performant dans la version modulo que dans la version classique. Cela signifie que les besoins en décalage sont plus importants que ceux en opérations locales. C'est ce rapport (temps_décalage / temps_operations_locales) qui détermine la meilleure technique d'adressage.

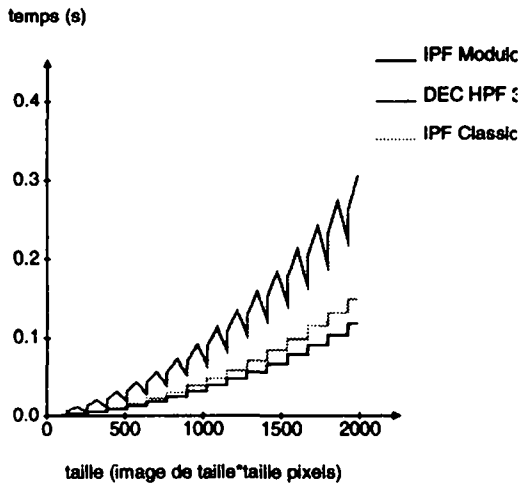


Figure 12 : résultat de la détection de contour avec le filtre de *Laplace*

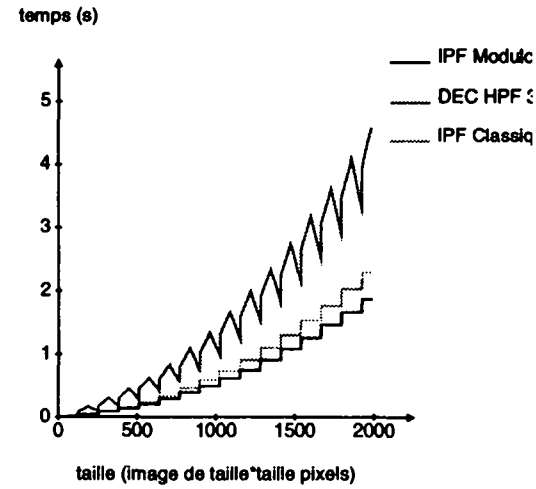


Figure 13 : résultat de la détection de contour avec les filtres de *Nevatia*

## 5.3 Rotation d'image

Il s'agit d'un algorithme de rotation d'image d'un angle quelconque compris entre -45° et +45°. Le traitement se décompose en deux phases distinctes, à savoir une phase de décalage horizontal et une phase de décalage vertical. On calcule pour cela des images de déplacement en fonction de l'angle et du centre de rotation.

Les résultats observés sur la Figure 14 confirment ceux obtenus pour la détection de contour. Ce type d'algorithme utilise intensément la primitive de décalage de tableau et tire avantageusement parti de la technique d'adressage modulo.

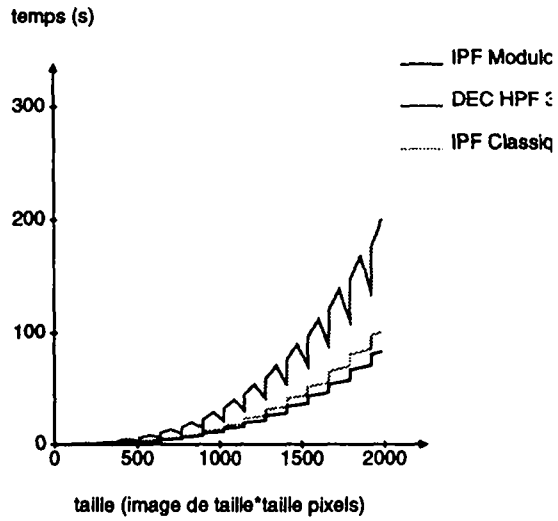


Figure 14 : résultat de la rotation d'image 2D d'entiers

## 5.4 Labellisation

Nous avons choisi de mettre en oeuvre deux algorithmes de labellisation d'image binaire. Le premier est l'algorithme de labellisation par *Broadcast* et le second l'algorithme de labellisation par *Shrinking*. Pour l'algorithme de *Shrinking*, qui se décompose en deux phases (réduction, expansion), seule la première phase a été implantée. La seconde s'appuie sur des expressions du type T(:,:,v), avec v valeur scalaire, déjà expérimentées dans la transformée de Hough.

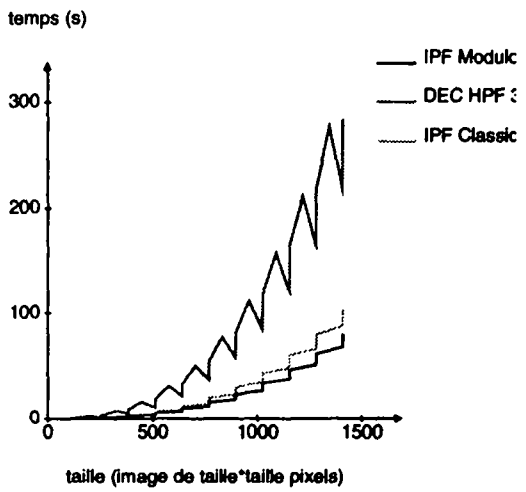


Figure 15 : résultat de la labellisation d'image 2D d'entiers par la méthode du *Broadcast*

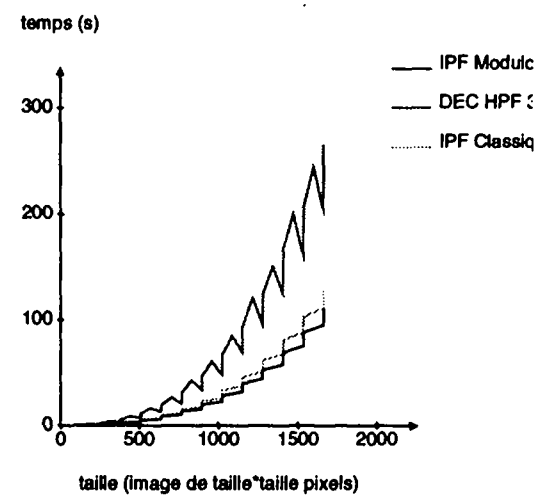


Figure 16 : résultat de la labellisation d'image 2D d'entiers par la méthode du *Shrinking*

Les temps présentés Figures 15 et 16 ne sont pas significatifs en absolu. Ils donnent seulement une indication des possibilités du langage par rapport à DEC HPF. Les performances de IPF sont ici encore supérieures à celles du compilateur DEC HPF.

## 5.5 Estimation de mouvement

Là encore, deux techniques d'estimation de mouvement local ont été testées. Le résultat est, à chaque fois, un champ de mouvement en chaque point. La méthode d'estimation différentielle de flot optique est basée sur le calcul des gradients horizontaux et verticaux en chaque point pour deux images successives. De ces gradients, en sont déduits les vecteurs déplacement dx et dy par un calcul itératif du flot optique. La méthode du *Block Matching* consiste à comparer une sous-bloc et une image. Une différence point à point est cumulée et le minimum extrait par réduction donne la nouvelle position la plus vraisemblable du sous-bloc dans l'image.

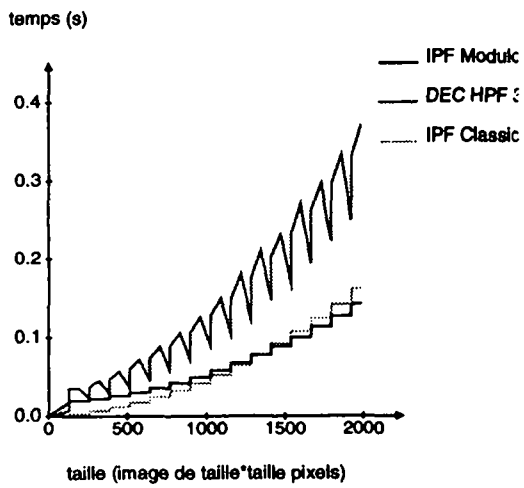


Figure 17 : résultat de l'estimation différentielle de flot optique

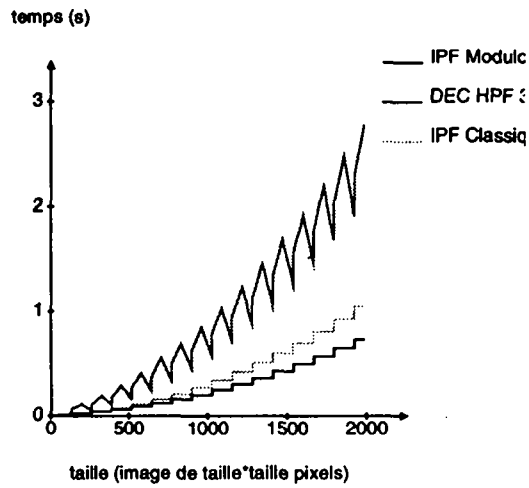


Figure 18 : résultat de l'estimation de mouvement selon la méthode du *Block Matching*

On peut remarquer que la technique du modulo donne de moins bonnes performances que la technique classique pour l'estimation de mouvement selon la méthode du *Gradient*. En dessous d'une certaine taille d'image, les temps de calcul sont plus coûteux que les opérations de décalage.

# 6 Conclusion et perspectives

Nous avons présenté dans ce rapport le langage IPF, élaboré dans le cadre du projet P$^3$I. IPF a été conçu pour répondre aux besoins de programmation d'applications de traitement d'image en temps réel vidéo sur le réseau SIMD de la machine P$^3$I. Sa syntaxe est tirée de Fortran 90 : les images sont modélisées sous forme de tableaux et manipulées comme des objets parallèles. Parmi les fonctionnalités de IPF, figure une nouvelle fonction intrinsèque de calcul de coordonnées cartésiennes, créée pour les besoins du domaine.

Un premier prototype de compilateur IPF a été développé sur DECmpp 12000 pendant la phase de mise au point de P$^3$I afin de valider les schémas de compilation développés. Parmi ces schémas, nous avons mis en oeuvre notamment la technique d'adressage modulo et l'allocation des données en mode bloc qui rendent les décalages d'image très performants, comme le montrent les résultats comparatifs avec le compilateur DEC HPF version 3.1. Les mesures de performance des opérations locales ont toutefois souligné que l'incrément modulo est quelque peu plus coûteux que l'incrément classique et qu'il serait rentable de l'implémenter au niveau du matériel, comme cela est fait dans les DSP (Digital Signal Processor) par exemple.

La gestion par le compilateur de la virtualisation des données permet d'envisager pour la suite des techniques d'optimisation telles que l'élimination de boucles inutiles et la fusion d'instructions parallèles. Nous allons également valider le module de génération de code assembleur pour l'unité SIMD de P$^3$I qui va se substituer au module de génération de MPL C du premier prototype. La phase d'optimisation comprendra, outre les techniques classiques telles que l'allocation des registres [Dietz 92] [Sabot 92] et celles évoquées plus haut, un module de compactage du micro-code assembleur de P$^3$I.

# 7 Bibliographie


[Albert 91] E. Albert, J.D. Lukas, G.L. Steele. **Data Parallel Computers and the FORALL Statement.** Journal of Parallel and Distributed Computing, 13, x, pp 185-192 (1991).

[Anandan 86] P. Anandan, L.R. Williams. **A coarse-to-fine control strategy for stereo and motion on a mesh-**

**connected computer.** Proceedings of the IEEE Conference on Conputer Vision and Pattern Recognition, pp 219-226 (1986).

[Bagrodia 90a] R. Bagrodia, K.M. Chandy, E. Kwan. **UC : A language for the Connection Machine.** Proceedings on Supercomputing'90, pp 525-534 (Novembre 1990).

[Becher 91] J.D. Becher, K.M. Hansen. **MPL : A Data Parallel C.** MasPar Computer Corporation (1991).

[Blelloch 90] G.E. Blelloch, S. Chatterjee. **VCODE : A data-parallel intermediate language.** Proceedings Frontiers'90 : The 3rd Symposium on the Frontiers of Massively Parallel Computation, pp 471-480 (Octobre 1990).

[Blelloch 91] G.E. Blelloch, J.M. Sipelstein. **Collection-Oriented Languages.** Proceedings of the IEEE, 79, 4, pp 504-521 (Avril 1991).

[Bouge 90] L. Bouge, P. Garda. **Towards a Semantic Approach to SIMD Architectures and their Languages.** Publication du LIENS, Paris (Fevrier 1990).

[Braun 91] T. Braunl. **Massively Parallel Programming with Parallaxis.** Publication interne, Université de Stuttgart, FRG (Decembre 1991).

[Busse 88] T. Busse. **MPP Pascal.** Proceedings Frontiers'88 : The 2nd Symposium on Frontiers of Massively Parallel Computation, pp 595-599 (Octobre 1988).

[Chaudhary 91] V. Chaudhary, J.K. Aggarwal. **On the Complexity of Parallel Image Component Labeling.** Proceedings of the 1991 International Conference on Paralle Processing, III, pp 183-187 (Août 1991).

[Chen 92] M. Chen, J. Cowie. **Prototyping Fortran-90 Compilers for Massively Parallel Machines.** Proceedings of the 1992 Conference on Programming Language Design & Implementation, 27, 7, pp 94-105 (Juillet 1992).

[Chen 93] L.T. Chen, L.S. Davis. **Parallel Curve Matching on the Connection Machine.** Pattern Recognition Lettres, 14, pp 133-140 (Février 1993).

[Christy 91] P. Christy. **Virtual Processors Considered Harmful.** pp 99-103 (1991).

[Cypher 87] R.E. Cypher, J.L.C. Sanz, L. Snyder. **The Hough Transform has O(N) Complexity on SIMD N by N Mesh Array Architectures.** Proceedings of the IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence, pp 115-121 (Octobre 1987).

[Cypher 90] R.E. Cypher, J.L.C. Sanz, L. Snyder. **Algorithms for Image Component Labeling on SIMD Mesh-Connected Computers.** IEEE Transactions on Computers, C-39, 2, pp 276-281 (Février 1990).

[DEC 92] Digital Equipment Corporation. **High Performance Fortran.** Reference Manual, (December 1992).

[Dietz 92] H.G. Dietz. **Common Subexpression Induction.** Proceedings of the 1992 International Conference

on Parallel Processing, II, pp 174-182 (Août 1992).

[Duff 86] M.J.B. Duff. **Intermediate-level Image Processing,** M.J.B. DUFF Editor, Academic Press (1986).

[Fang 89] Z. Fang, X. Li, L.M. Ni. **On the Communication Complexity of Generalized 2-D Convolution on Array Processors.** IEEE Transactions on Computers, C-38, 2, pp 184-193 (Février 1989).

[Gannon 91] D. Gannon, J.K. Lee. **Object-Oriented Parallelism : PC++ Ideas and Experiments.** Email : Report, Department of Computer Science, Indiana, pp 13-23 (1991).

[Gannon 92] D. Gannon, J.K. Lee, B. Shei, S. Sarukaï, S. Narayana, N. Sundresan, D. Atapattu, F. Bodin. **Sigma II : A tool kit for building parallelizing compilers and performance analysis systems.** Elsevier (1992).

[Guerra 87] C. Guerra, S. Hambrusch. **Parallel algorithms for line detection on a mesh.** Proceedings of the IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence, pp 99-106 (Octobre 1987).

[Hambrusch 89] S. Hambrusch, L. Tewinkel. **A study of Connected Component Labeling Algorithms on the MPP.** pp 477-483 (1989).

[Heitz 90] F. Heitz, P. Bouthemy. **Multimodal motion estimation and segmentation using Markov random fields.** Proceedings of the 10th International Conference on Pattern Recognition, pp 378-383 (Juin 1990).

[Herbordt 91] M.C. Herbordt, C.C. Weems, M.J. Scudder. **A Computational Framework and SIMD Algorithms for Low-Level of Intermediate Level Vision Processing.** Proceedings of the 1991 International Conference on Computer Vision and Pattern Recognition, pp 740-741 (Juin 1991).

[Holey 92] J.A. Holey, O.H. Ibarra. **Iterative Algorithms for the planar Convex Hull Problem on Mesh-Connected Arrays.** Parallel Computing, 18, pp 281-296 (1992).

[Hsu 90] W.J. Hsu, L.R. Wu, X. Lin. **Optimal algorithms for labeling image components.** Proceedings of the 1990 International Conference on Parallel Processing, III, pp 75-82 (Août 1990).

[Ibrahim 85] H.A.H. Ibrahim, J.R. Kender, D.E. Shaw. **The Analysis and Performance of Two Middle-Level Vision Tasks on a Fine-Grained SIMD Tree Machine.** Proceedings of the 1985 Conference on Computer Vision and Pattern Recognition, pp 248-256 (1985).

[Jacobsen 90a] K.P. Jacobsen, C. Kuszmaul. **Small Fast Fourier Transforms.** Application note, MasPar Computer Corporation (1990).

[Jacobsen 90b] K.P. Jacobsen. **Image Convolutions.** Application note, MasPar Computer Corporation (1990).

[Jamieson 86] L.H. Jamieson, P.T. Mueller, H.J. Siegel. **FFT Algorithms for SIMD Parallel Processing Systems.** Journal of Parallel and Distributed Computing, 3, pp 48-71 (Mars 1986).

[Kannan 89] C.S. Kannan, H.Y.H. Chuang. **Fast Hough transform on a mesh-connected processor array.**

Information Processing Letters, 33, pp 243-248 (1989).

[Knobe 88] K. Knobe, J.D. Lukas, G.L. Steele. **Massively parallel data optimization.** Proceedings Frontiers'88 : The 2nd Symposium on the Frontiers of Massively Parallel Computation, pp 551-558 (Octobre 1988).

[Knobe 90] K. Knobe, J.D. Lukas, G.L. Steele. **Data optimization : Allocation of arrays to reduce communication on SIMD machines.** Journal of Parallel and Distributed Computing, 8, 2, pp 102-118 (Fevrier 1990).

[Kuehn 85] J.T. Kuehn, H.J. Siegel. **Extensions to the C programming language for SIMD/MIMD parallelism.** Proceedings of the 1985 International Conference on Parallel Processing, II, pp 232-235 (Août 1985).

[Li 93] Z.N. Li, B. Yao, F. Tong. **Linear Generalized Hough Transform and its Parallelization.** Image and Vision Computing, 11, 1, pp 11-24 (Janvier 1993).

[Manohar 89] M. Manohar, H.K. Ramapriyan. **Connected Component Labeling of Binary Images on a Mesh-Connected Massively Parallel Processor.** Computer Vision, Graphics, and Image Processing, 45, pp 133-149 (1989).

[Mémin 92] E. Mémin, F. Heitz. **Algorithmes parallèles pour l'analyse d'image par champs markoviens.** Publication interne IRISA no 657 (Mai 1992).

[Metcalf 90] M. Metcalf, J. Reid. **Fortran 90 explained.** Oxford University Press (1990).

[Miller 85] R. Miller. **Writing SIMD algorithms.** Proceedings of the 1985 International Conference on Computer Design : VLSI in Computers, pp 122-125 (1985).

[Miller 91] R. Miller, S.T. Tanimoto. **Detecting Repeated Patterns on Mesh Computers.** Proceedings of the 1991 International Conference on Parallel Processing, III, pp 188-191 (Août 1991).

[Mou 90] Z.G. Mou. **Divacon : A Parallel Language for Scientific Computing Based on Divide-and-Conquer.** Proceedings Frontiers'90 : The 3rd Symposium on the Frontiers of Massively Parallel Computation, pp 451-461 (Octobre 1990).

[Nassimi 80] D. Nassimi, S. Sahni. **Finding connected components and connected ones on a mesh-connected parallel computer.** SIAM Journal of Computing, 9, 4, pp 744-757 (Novembre 1980).

[Noble 88] J.A. Noble. **Finding Corners.** Image and Vision Computing, 6, 2, pp 121-128 (Mai 1988).

[Paris 88] N. Paris. **Définition du langage POMPC.** Publication interne, CNRS (Décembre 1988).

[Philippsen 91] M. Philippsen, W.F. Tichy. **Compiling for Massively Parallel Machines.** Publication interne, Karlsruhe University, FRG (1991).

[Quinn 91] M.J. Quinn, P.J. Hatcher, R.J. Anderson, A.J. Lapadula, B.K. Seevers, A.F. Bennett. **Architecture-Independent Scientific Programming in Dataparallel C : Tree Case Studies.** Proceedings of

the 1991 International Conference on Supercomputing, pp 208-217 (Novembre 1991).

[Reeves 84] A.P. Reeves. **Parallel Pascal : An Extended Pascal for Parallel Computers.** Journal of Parallel and Distributed Computing, 1, pp 64-80 (Janvier 1984).

[Reeves 85] A.P. Reeves, C.H. Francfort. **Data mapping and rotation functions for the massively parallel processor.** Proceedings of the 1985 Workshop on Computer Architecture Pattern Analysis Image Database Management, pp 412-419 (Novembre 1985).

[Reeves 90] A.P. Reeves, M. Willebeek-Lemair. **Solving Nonuniform Problems on SIMD Computers : Case Study on Region Growing.** Journal of Parallel and Distributed Computing, 8, pp 135-149 (1990)

[Rice 88] M.D. Rice, S.B. Seidman, P.Y. Wang. **A High-Level Language for SIMD Computation.** Proceedings of the 1988 Conference on Algorithms and Hardware for Parallel Processing, pp 385-391 (1988).

[Sabot 88] G.W. Sabot. **The Paralation Model : Architecture-Independent Parallel Programming.** The MIT Press (1988).

[Sabot 92] G.W. Sabot. **Optimized CM Fortran Compiler for the Connection Machine Computer.** Proceedings of the 25th Hawaii International Conference on System Sciences, II, pp 161-172 (1992).

[Steele 86] G.H. Steele, W.D. Hillis. **Connection Machine Lisp : Fine-Grain Parallel Symbolic Processing.** Proceedings of the 1986 Conference on Lisp and Functional Programming, pp 279-297 (Août 1986).

[Steele 87] G.L. Steele, J.R. Rose. **C* : An extended C language for data parallel programming.** Proceedings of the 2nd International Conference on Supercomputing, pp 2-16 (Mai 1987).

[Strong 82] J.P. Strong. **Basic Image Processing Algorithms on the Massively Parallel Processor.** Multicomputers and Image Processing Algorithms and Programs, K. PRESTON, Jr. and L. Hur Editors, Academic Press (1982).

[Strong 91] J.P. Strong. **Computations on the Massively Parallel Processor at the Goddard Space Flight Center.** Proceedings of the IEEE, 79, 4, pp 548-558 (Avril 1991).

[Tilton 88] J.C. Tilton. **Image Segmentation by Iterative Parallel Region Growing with Applications to Data Compression and Image Analysis.** Proceedings Frontiers'88 : The 2rd Symposium on the Frontiers of Massively Parallel Computation, pp 357-360 (Octobre 1988).

[Tong 91] C. Tong, P.N. Swarztrauber. **Ordered Fast Fourier Transforms on a Massively Parallel Hypercube Multiprocessor.** Journal of Parallel and Distributed Computing, 12, pp 50-59 (1991).

[Warpenburg 82] M.R. Warpenburg, L.J. Siegel. **Image resampling in an SIMD environment.** IEEE Transactions on Computers, 31, 10, pp 934-942 (Octobre 1982).

[Webb 92] J.A. Webb. **Steps Toward Architecture-Independent Image Processing.** Computer, 2, pp 21-30

(Février 1992).

[Weems 91] C.C. Weems. **Architectural requirements of image understanding with respect to parallel processing.** Proceedings of the IEEE, 79, 4 (April 1991).

[Weiss 91] M. Weiss. **Strip Mining on SIMD Architectures.** Proceedings of the 1991 International Conference on Supercomputing, (Juin 1991).

[Williams 86] L.R. Williams, P. Anandan. **A Coarse-to-Fine Control Strategy for Stereo and Motion on a Mesh-Connected Computer.** Proceedings of the 1991 International Conference on Computer Vision and Pattern Recognition, pp 219-226 (1986).

[Zerubia 92] J. Zerubia, F. Ployette. **Détection de Contours et Lissage d'Image par Deux Algorithmes Déterministes de Relaxation. Mise en Oeuvre sur la Machine à Connexions CM2.** Traitement du Signal, 8, 3, pp 165-179 (1992).

[Zima 91] H. Zima, B. Chapman. **Supercompilers for Parallel and Vector Computers.** The ACM Press (1991).

Unité de Recherche INRIA Rennes
IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399